1 Цель

Применить стохастический спуск для решения задачи линейной регрессии и проанализировать, как изменение размера батча влияет на сходимость алгоритма.

Постановка задачи

1. Реализуйте стохастический градиентный спуск для решения линейной регрессии. Исследуйте сходимость с разным размером батча (1 - SGD, 2, ..., $n-1$ - Minibatch GD, n - GD из предыдущей работы).

2. Подберите функцию изменения шага (learning rate scheduling), чтобы улучшить сходимость, например экспоненциальную или ступенчатую.

3. Исследуйте модификации градиентного спуска (Nesterov, Momentum, AdaGrad, RMSProp, Adam).

4. Исследуйте сходимость алгоритмов. Сравнить различные методы по скорости сходимости, надежности, требуемым машинным ресурсам (объем оперативной памяти, количеству арифметических операций, времени выполнения)

5. Постройте траекторию спуска различных алгоритмов из одной и той же исходной точки с одинаковой точностью. В отчете наложить эту траекторию на рисунок с линиями равного уровня заданной функции.

6. Реализуйте полиномиальную регрессию. Постройте графики восстановленной регрессии для полиномов разной степени.

7. Модифицируйте полиномиальную регрессию добавлением регуляризации в модель (L1, L2, Elastic регуляризации).

8. Исследуйте влияние регуляризации на восстановление регрессии.

2 Решение

1. Алгоритм стохастического спуска с подвижным батчем.

```cpp
qreal mse(v<pr<qreal, qreal>> const& points, qreal const k, qreal const b) noexcept {
    auto f = [&k, &b](qreal const x) {
        return k * x + b;
    };

    qreal result = 0, diff;

    for (auto const& elem : points) {
        diff = elem.second - f(elem.first);
        result += diff * diff;
    }
    return result / points.size();
}

v<int> rand_seq(int const k, int const n) {...}

std::tuple<qreal, qreal, int> linear_regression(const way_t &points, const int batch, const qreal lrk, const qreal lrb, const qreal k, const qreal b, const int max_step, const qreal dlt)
{
    assert(batch > 0 && batch <= points.size());
    qreal optimal_sse = mse(points, k, b);
    pr<qreal, qreal> cur = {0, 0};
    qreal cur_sse;

    for (int i = 1; i <= max_step; ++i) {
        cur = step(points, cur.first, cur.second, lrk, lrb, batch);
        if (std::isnan(cur.first) || std::isnan(cur.second)) {
            return {cur.first, cur.second, i};
        }
#if DEBUG_OUTPUT
        if (i % 10 == 0) {
            qDebug() << cur.first << cur.second;
        }
#endif
        cur_sse = mse(points, cur.first, cur.second);

        if (std::abs(optimal_sse - cur_sse) < dlt) {
            return {cur.first, cur.second, i};
        }
    }

    return {cur.first, cur.second, max_step};
}

pr<qreal, qreal> step(const way_t &points, qreal const k, qreal const b, qreal const ck, qreal const cb, int const batch) {
    auto chosen_index = rand_seq(batch, points.size());

    qreal gradk = 0;
    qreal gradb = 0;
    qreal temp;

    for (auto const& elem : chosen_index) {
        temp = points[elem].second - (points[elem].first * k) - b;
        gradk += (-2. / points.size()) * temp * points[elem].first;
        gradb += (-2. / points.size()) * temp;
    }

    return {k - ck * gradk, b - cb * gradb};
}
```

Первая функция высчитывающая MSE по всем точкам и k и b.

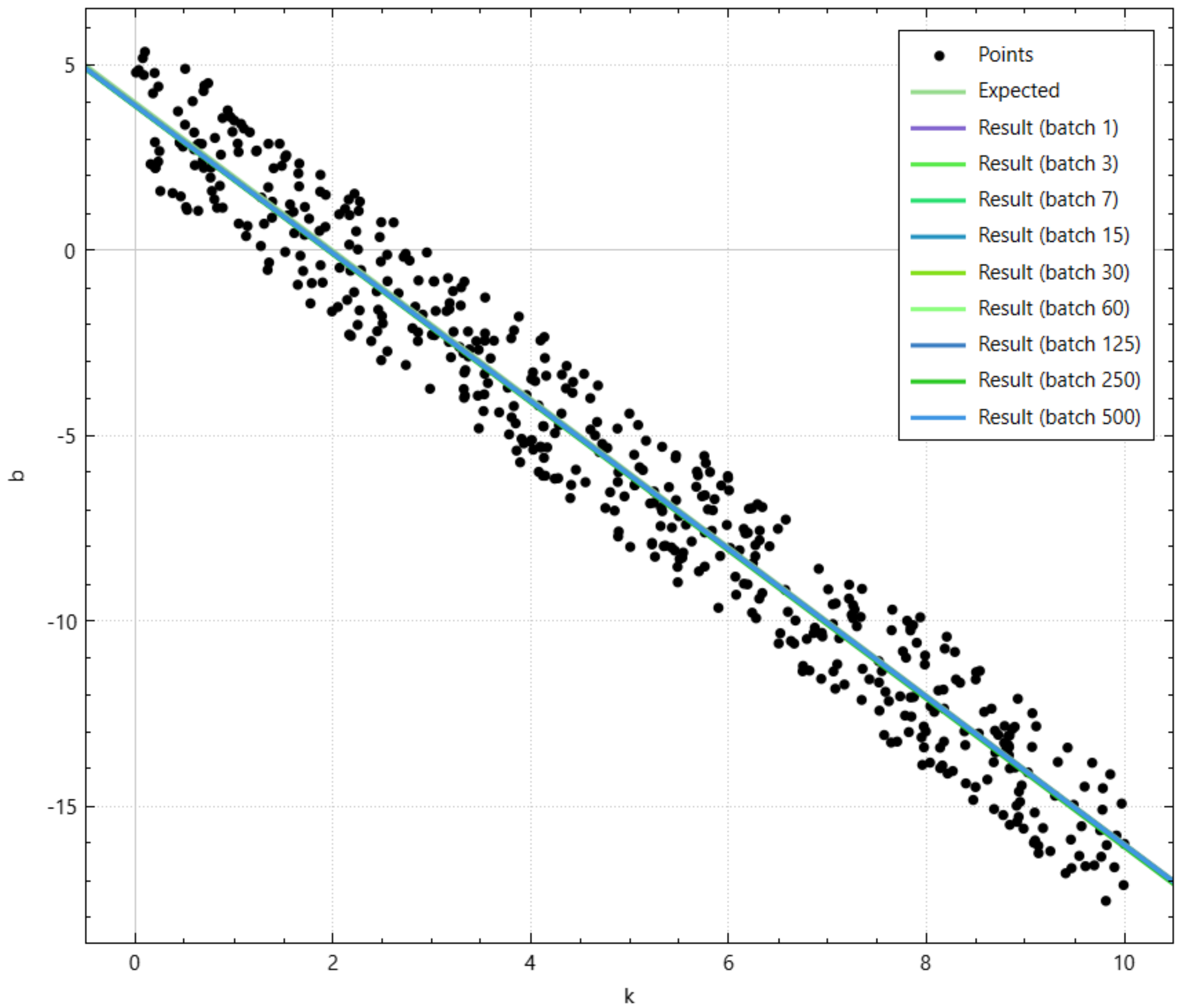
Вторая функция даёт радномную последовательность из неповторяющихся чисел. Необходимо для взятия определённого подмножества точек, на итерации стохастического спуска по определённым точкам.

Третья функция линейной регрессии возвращающая k, b количество шагов необходимых для достижения ответа.

Функция высчитывает градиент по параметрам k и b функции MSE среднеквадратичного расстояния прямой до точек нашего множества и отправляет новые значения k и b.

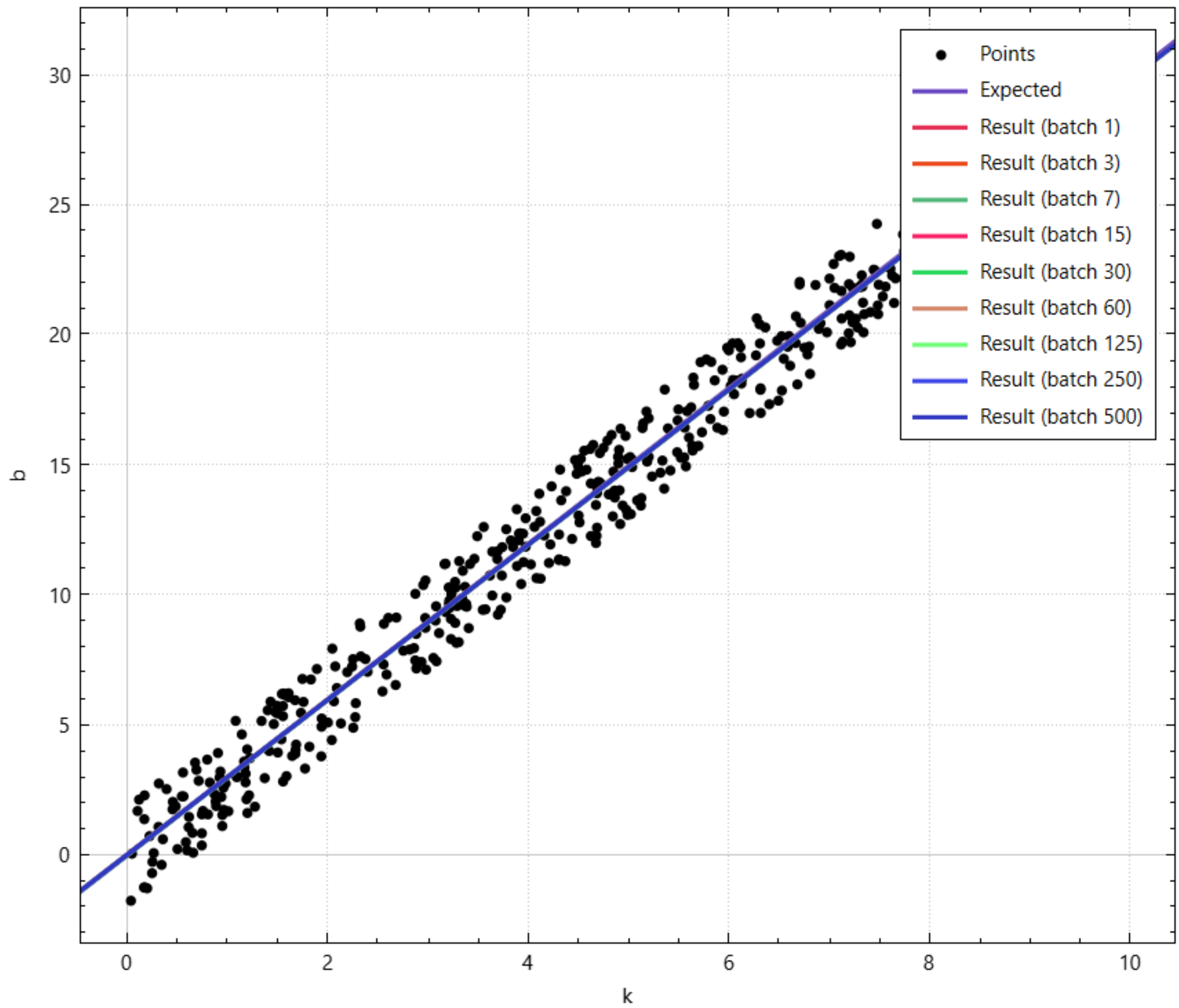
Теперь рассмотрим 3 функции по которым мы будем сравнивать эффективность и результативность линейной регрессии с разным размером батча:

1.1. $y = -2x + 4$



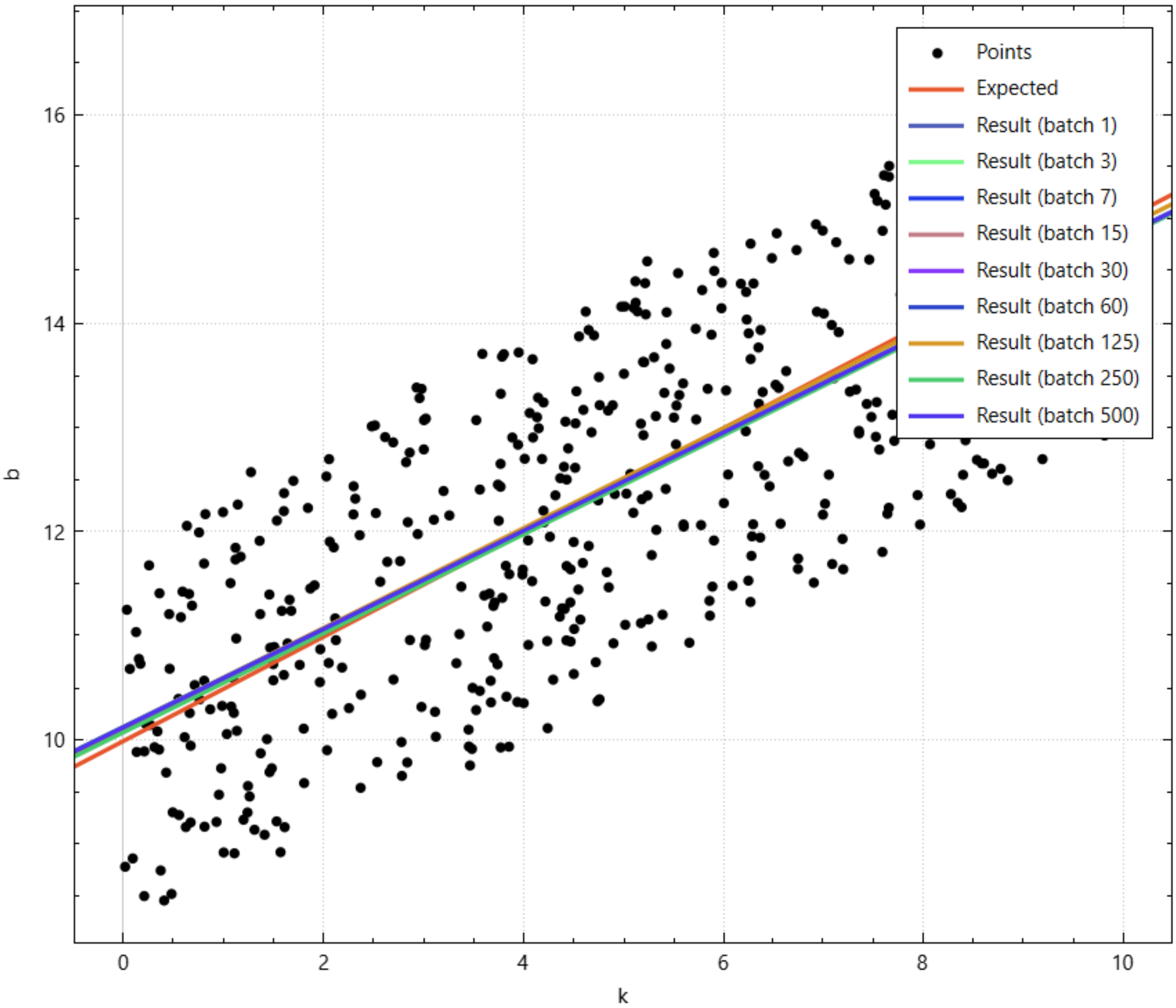
Batch	St	Result function
1	39299	$y = -2.00354x + 3.98835$
3	14603	$y = -1.96315x + 3.80778$
7	6400	$y = -1.99796x + 3.96357$
15	2954	$y = -2.00008x + 4.00181$
30	1421	$y = -1.97652x + 3.88523$
60	821	$y = -1.99487x + 3.94875$
125	318	$y = -2.00876x + 4.00058$
250	191	$y = -2.00049x + 4.02615$
500	91	$y = -1.97254x + 3.83978$

1.2. $y = 3x$



Batch	Steps	Result function
1	46921	$y = 2.99833x + -0.0488231$
3	16089	$y = 2.97525x + 0.170553$
7	6693	$y = 2.97205x + 0.0880841$
15	2480	$y = 2.95908x + 0.129934$
30	1526	$y = 2.97846x + 0.148212$
60	639	$y = -1.99487x + 3.94875$
125	369	$y = 2.95654x + 0.17973$
250	194	$y = 2.99714x + 0.00517378$
500	69	$y = 2.93625x + 0.402302$

1.3. $y = \frac{1}{2}x + 10$



Batch	Steps	Result function
1	9152	$y = 2.99833x + -0.0488231$
3	3041	$y = 2.97525x + 0.170553$
7	1346	$y = 2.97205x + 0.0880841$
15	890	$y = 2.95908x + 0.129934$
30	374	$y = 2.97846x + 0.148212$
60	175	$y = -1.99487x + 3.94875$
125	68	$y = 2.95654x + 0.17973$
250	30	$y = 2.99714x + 0.00517378$
500	42	$y = 2.93625x + 0.402302$

Из полученных результатов можно сделать как минимум два вывода.

Во-первых, при меньшем батче получается большее количество ходов. Оно и понятно, потому что за один ход с меньшим батчем мы не только делаем меньший сдвиг к нашему решению, но и вообще можем с большей погрешностью идти к решению, так как мы берём подмножество точек, которое хоть и достаточко для сдвига, но чем их меньше тем больше размазана картина.

Во-вторых, степерь дальности нашего стартого коэфициента k от решения даёт гораздо большее количество итерации нежели при токой же степени дальности параметра b. Могу объяснить тем, что хоть scaling параметра k даёт выигрыш, но всё равно его сложно подобрать в зависимости от примера.

Ну а в итоге на точность батч видимым образом не влияет.

2. Мы можем выбрать иную функцию для изменения шага в алгоритме. Важно отметить, что в предыдущей версии алгоритма мы каждый раз пересчитывали функционал качества Q и сравнивали его с ожидаемым значением Q.

   Чтобы ускорить алгоритм SGD, мы можем использовать формулу экспоненциального скользящего среднего (EMA). На каждой итерации мы будем пересчитывать функционалы качества Q и Q_0 с использованием функции $\mathrm{MSE} = \lambda\Delta(y_i - f(x_i))^2 + (1-\lambda)\mathrm{MSE}_{\mathrm{prev}}$ и сравнивать их с помощью модуля и дельты.

   При сравнении результатов запуска градиентного спуска на наших функциях, мы получаем впечатляющие результаты. При использовании EMA для пересчета функционала качества мы достигаем значительного выигрыша во времени, в несколько тысяч раз. Это особенно заметно на больших наборах данных, что подчеркивает важность данного метода.

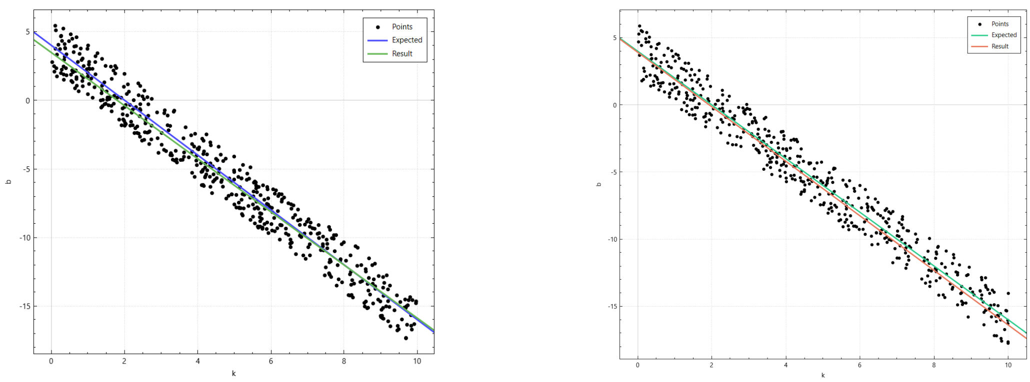
Код метода:

```cpp
std::tuple<qreal, qreal, int> sdg_linear_regression(
    const way_t &points,
    const qreal lrk,
    const qreal lrb,
    const qreal k,
    const qreal b,
    const int max_step,
    const qreal dlt)
{
    qreal mse_bs = mse(points, k, b);
    qreal cur_mse = 0;
    qreal const coef = 2. / (points.size() + 1);
    pr<qreal, qreal> cur = {0, 0};
    qreal diff;
    auto f = [&cur](qreal const x) {
        return cur.first * x + cur.second;
    };
    auto x = [&points](int const i) { return points[i % points.size()].first;};
    auto y = [&points](int const i) { return points[i % points.size()].second;};

    for (int i = 0; i < max_step; ++i) {
        diff = y(i) - f(x(i));
        cur.first -= lrk  * (-2.) * diff * x(i);
        cur.second -= lrb  * (-2.) * diff;
        diff *= diff;

        mse_bs = coef * diff + (1 - coef) * mse_bs;
        cur_mse = coef * diff + (1 - coef) * cur_mse;

#if DEBUG_OUTPUT
        if (i % 100 == 0) {
            qDebug() << mse_bs << cur_mse << std::abs(mse_bs - cur_mse);
        }
#endif

        if (std::abs(mse_bs - cur_mse) < dlt) {
            return {cur.first, cur.second, i};
        }
    }
    return {cur.first, cur.second, max_step};
}
```

Теперь покажем примеры результатов разных алгоритмов на наших вышеупомянутых функциях.

2.1. $y = -2x + 4$

Рис. 1

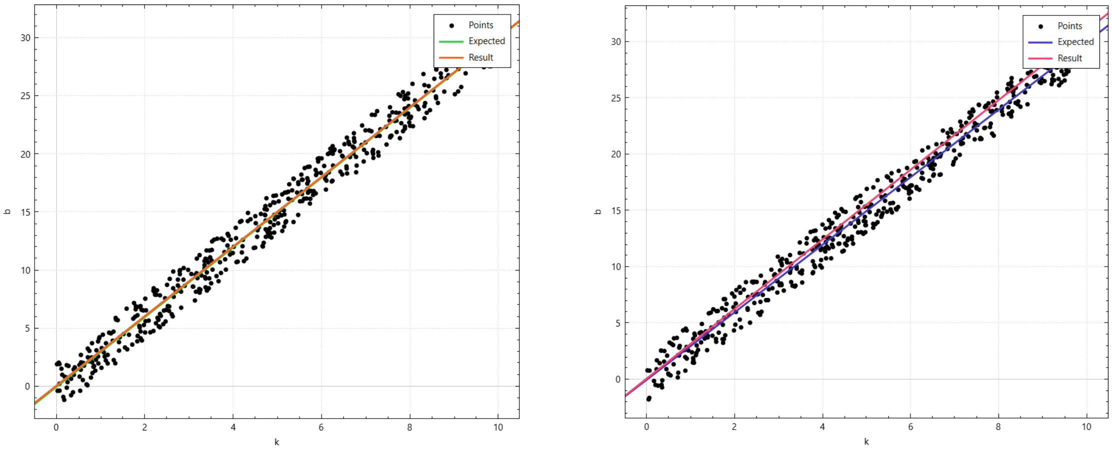


(a) Первый алгоритм с батчем 1. Ходов: 90075. Время: 2798 ms

(b) Стохастический с оптимизацией. Ходов: 2369. Время: 4 ms

2.2. $y = 3x$

Рис. 2

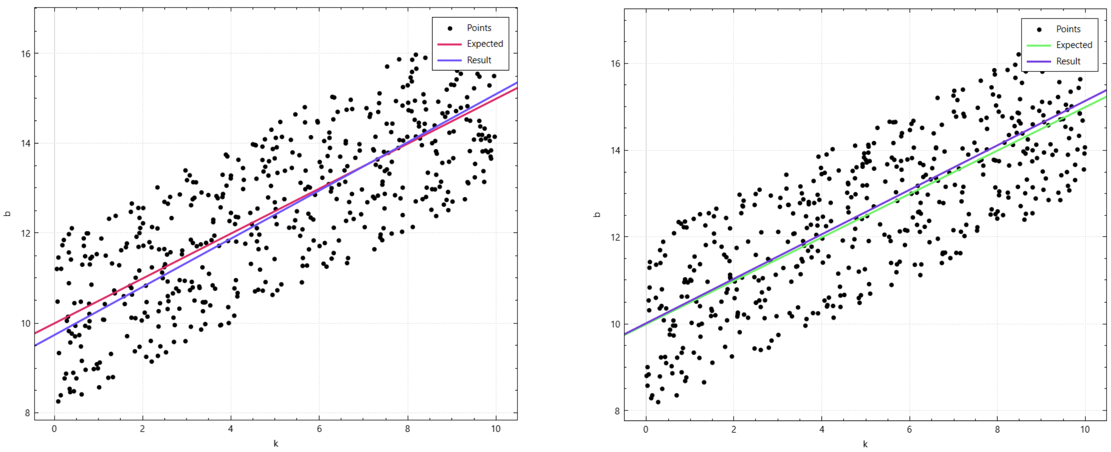


(a) Первый алгоритм с батчем 1.
Ходов: 83047. Время: 2593 ms

(b) Стохастический с оптимизацией.
Ходов: 2367. Время: 2 ms

2.3. $y = \frac{1}{2}x + 10$

Рис. 3



(a) Первый алгоритм с батчем 1.
Ходов: 100000. Время: 4131 ms

(b) Стохастический с оптимизацией.
Ходов: 2370. Время: 2 ms

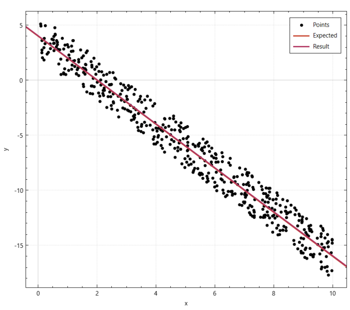
3. 3.1. Для начала рассмотрим метод **Momentum**. Он представляет собой эффективное решение проблемы, с которой сталкивается SGD (стохастический градиентный спуск) при оптимизации функций, где значение функции меняется значительно быстрее по одной из переменных, чем в других

направлениях. Этот метод также помогает преодолеть проблему попадания в локальные минимумы, предоставляя больше возможностей для выхода из них. Мы вводим понятие момента ΔW_N, который со временем накапливает влияние весов предыдущих градиентов.

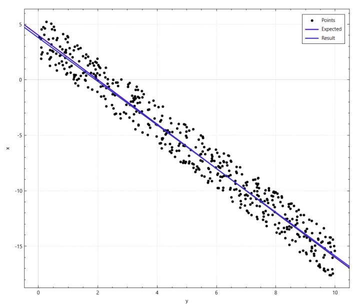
Вот его код:

```
v<QCPCurveData> momentum_linear_regression(
    const way_t &points,
    const qreal lrk,
    const qreal lrb,
    const qreal k,
    const qreal b,
    const int max_step,
    const qreal dlt)
{
    qreal const m_force = 0.5;
    qreal const optimal_mse = mse(points, k, b);

    qreal b_force = 0;
    qreal cur_mse;
    qreal diff;
    qreal k_force = 0;
    qreal gradk;
    qreal gradb;
    pr<qreal, qreal> cur = {0, 0};

    auto f = [&cur](qreal const x) {
        return cur.first * x + cur.second;
    };
    auto x = [&points](int const i) { return points[i % points.size()].first;};
    auto y = [&points](int const i) { return points[i % points.size()].second;};

    v<QCPCurveData> way = {{0, cur.first, cur.second}};

    for (int i = 1; i <= max_step; ++i) {
        diff = y(i) - f(x(i));
        gradk = -2. * diff * x(i);
        gradb = -2. * diff;
        k_force = m_force * k_force - lrk * gradk;
        b_force = m_force * b_force - lrb * gradb;
        cur.first += k_force;
        cur.second += b_force;

        cur_mse = mse(points, cur.first, cur.second);

        way.emplace_back(i, cur.first, cur.second);
        if (std::abs(cur_mse - optimal_mse) < dlt) {
            break;
        }
    }
    return way;
}
```

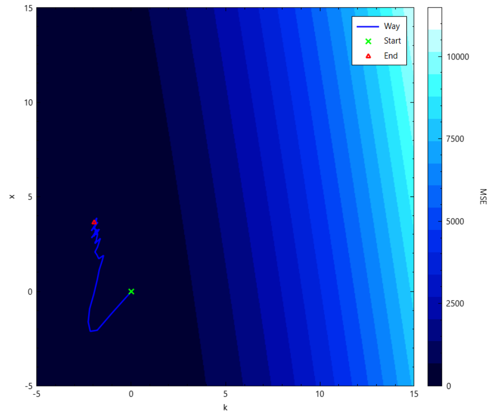
i. $y = -2x + 4$



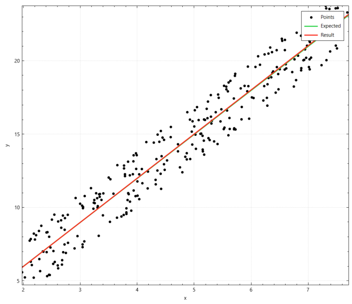
(a) Momentum. Ходов: 198. Время: 9 ms



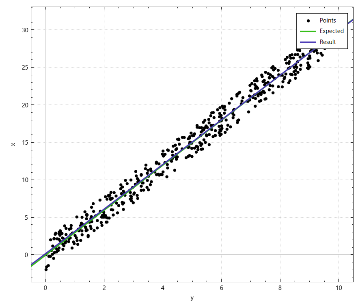
(b) Стохастический без оптимизацией. Ходов: 100000. Время: 8993 ms



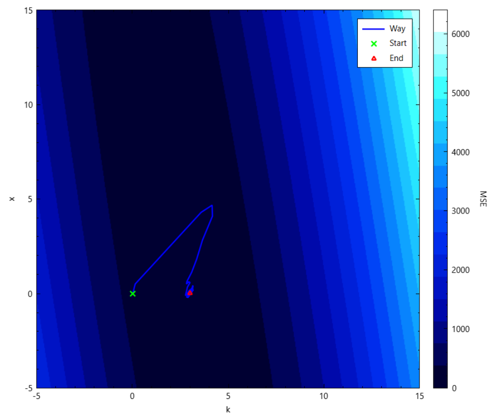
ii. $y = 3x$



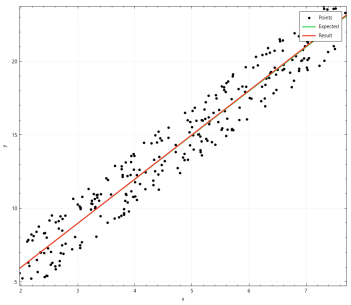
(a) Momentum. Ходов: 104. Время: 3 ms



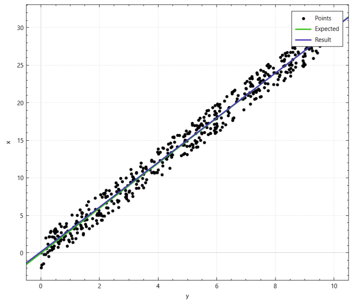
(b) Стохастический без оптимизацией. Ходов: 83665. Время: 3799 ms



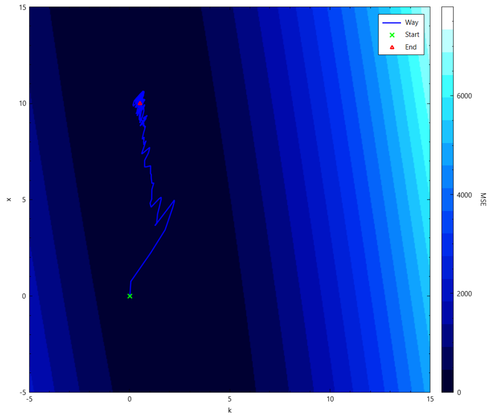
iii. $y = \frac{1}{2}x + 10$



(a) Momentum. Ходов: 122. Время: 3 ms



(b) Стохастический без оптимизацией. Ходов: 100000. Время: 4779 ms



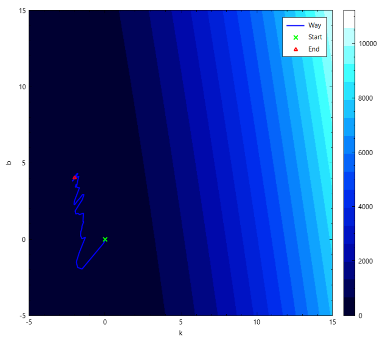
Мы заметили значительное улучшение скорости сходимости и существенное сокращение времени работы при использовании метода Momentum в процессе оптимизации. Этот метод значительно повысил эффективность алгоритма и привел к существенному ускорению общего процесса.

3.2. **Nesterov**. Давайте применим технику сглаживания нашего градиентного спуска. Мы можем использовать информацию о предыдущем импульсе, чтобы вычислить градиент в определенной точке. Это поможет нам получить более гладкий путь к точке минимума и улучшить стабильность и эффективность нашего градиентного спуска.

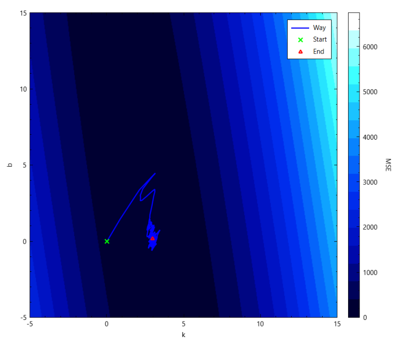
i. $y = -2x + 4$

ii. $y = 3x$

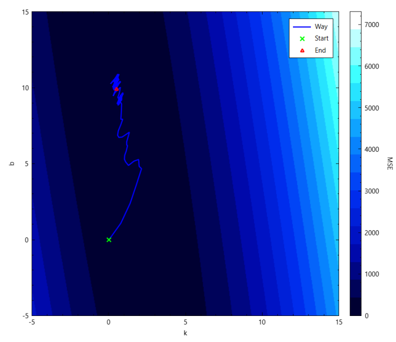
iii. $y = \frac{1}{2}x + 10$



(a) Ходов: 65. Время: 1 ms



(b) Ходов: 344. Время: 5 ms



(c) Ходов: 154. Время: 3 ms

Сравнивая этот метод с предыдущим, можно сразу заметить что траектория пути стала более сглаженной.

И кстати, код:

```
v<QCPCurveData> nesterov_linear_regression(const way_t &points, const qreal lrk, const qreal lrb, const qreal k, const qreal b, const int max_step,
{
    qreal const m_force = 0.6;
    qreal const optimal_mse = mse(points, k, b);

    qreal cur_mse;
    qreal b_force = 0;
    qreal k_force = 0;
    qreal gradk;
    qreal gradb;

    pr<qreal, qreal> cur = {0, 0};
    auto f = [&cur, &lrk, &k_force, &lrb, &b_force](qreal const x) {
        return (cur.first - lrk * k_force) * x + cur.second - lrb * b_force;
    };
    auto x = [&points](int const i) { return points[i % points.size()].first;};
    auto y = [&points](int const i) { return points[i % points.size()].second;};

    v<QCPCurveData> way = {{0, cur.first, cur.second}};

    for (int i = 0; i < max_step; ++i) {
        gradk = -2. * (y(i) - f(x(i))) * x(i);
        gradb = -2. * (y(i) - f(x(i)));
        k_force = m_force * k_force - lrk * gradk;
        b_force = m_force * b_force - lrb * gradb;
        cur.first += k_force;
        cur.second += b_force;

        cur_mse = mse(points, cur.first, cur.second);
        way.emplace_back(i, cur.first, cur.second);
        if (std::abs(cur_mse - optimal_mse) < dlt) {
            break;
        }
    }
    return way;
}
```

3.3. **AdaGrad**. Давайте рассмотрим другой подход. Мы можем воспользоваться идеей смещения на среднее значение градиента, чтобы учесть все итерации и достичь сбалансированной выборки. Таким образом, мы учитываем влияние всех промежуточных шагов и обеспечиваем равновесие в процессе обучения.

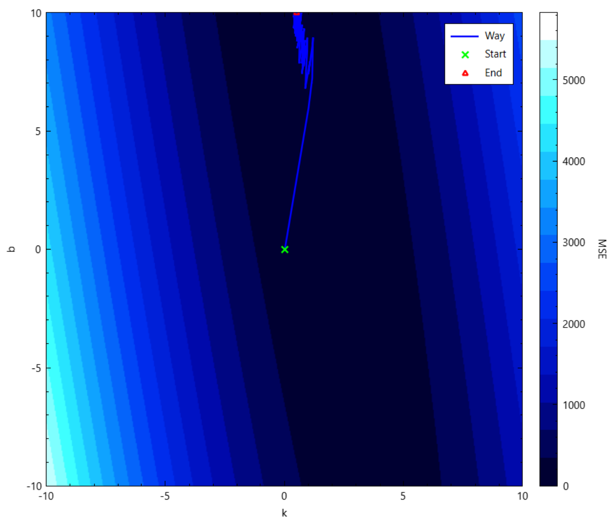
Кроме того, стоит упомянуть проблему, которая требует внедрения метода RMSProp. В этом случае возникает проблема быстрого роста знаменателя, который накапливает сумму квадратов градиентов. Это может привести к замедлению скорости обучения, делая ее практически незаметной, и, в конечном итоге, лишая алгоритм возможности эффективно обучаться.

Ниже представлен код:

```cpp
v<QCPCurveData> adagrad_linear_regression(
    const way_t &points,
    qreal lrk,
    qreal lrb,
    const qreal k,
    const qreal b,
    const int max_step,
    const qreal dlt)
{
    qreal const optimal_mse = mse(points, k, b);
    lrk *= 250;
    lrb *= 150;
    qreal gk = 0;
    qreal gb = 0;
    qreal gradk;
    qreal gradb;
    qreal diff;
    qreal cur_mse;

    pr<qreal, qreal> cur = {0, 0};
    auto f = [&cur](qreal const x) {
        return cur.first * x + cur.second;
    };
    auto x = [&points](int const i) { return points[i % points.size()].first;};
    auto y = [&points](int const i) { return points[i % points.size()].second;};

    v<QCPCurveData> way = {{-1, cur.first, cur.second}};

    for (int i = 0; i < max_step; ++i) {
        diff = y(i) - f(x(i));
        gradk = -2. * diff * x(i);
        gradb = -2. * diff;
        gk += gradk * gradk;
        gb += gradb * gradb;
        cur.first -= (lrk / std::sqrt(gk + dlt)) * gradk;
        cur.second -= (lrb / std::sqrt(gb + dlt)) * gradb;

        cur_mse = mse(points, cur.first, cur.second);

        way.emplace_back(i, cur.first, cur.second);
        if (std::abs(cur_mse - optimal_mse) < dlt) {
            break;
        }
    }

    return way;
}
```

Тест на прямой $y = \frac{1}{2}x + 10$

Рис. 8. Ходов: 797. Время: 5 ms



На этом примере мы можем увидеть, что траектория поиска решения стала более прямой и не расплывчатой.

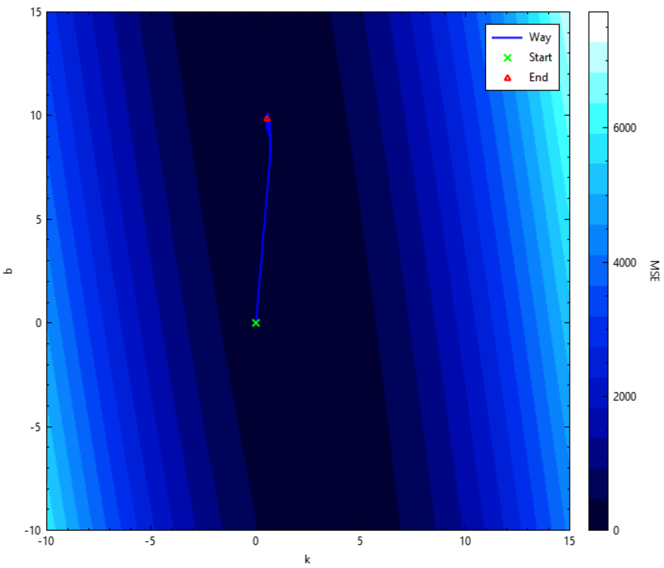
3.4. **RMSProp** В данном алгоритме происходит замена суммы квадратов градиентов на экспоненциально затухающее среднее всех предыдущих квадратов градиентов. Это означает, что вместо простого суммирования всех квадратов градиентов, мы учитываем значения градиентов из более поздних точек более существенным образом. Экспоненциальное затухание означает, что чем более поздние точки, тем меньше их вклад в общее значение. Таким образом, этот подход сосредоточивается на последних значениях частных производных и дает им больший вес в алгоритме.

Ниже представлен код:

```cpp
v<QCPCurveData> rmsprop_linear_regression(const way_t &points, qreal lrk, qreal lrb, const qreal k, const qreal b, const int max_step, const qreal dlt)
{
    qreal const optimal_mse = mse(points, k, b);
    qreal const pwr = 0.9;
    qreal gk = 0;
    qreal gb = 0;
    qreal diff;
    qreal gradk;
    qreal gradb;
    qreal cur_mse;

    pr<qreal, qreal> cur = {0, 0};
    auto f = [&cur](qreal const x) {
        return cur.first * x + cur.second;
    };
    auto x = [&points](int const i) { return points[i % points.size()].first;};
    auto y = [&points](int const i) { return points[i % points.size()].second;};

    v<QCPCurveData> way = {{-1, cur.first, cur.second}};

    for (int i = 0; i < max_step; ++i) {
        diff = y(i) - f(x(i));
        gradk = -2. * diff * x(i);
        gradb = -2. * diff;
        gk = pwr * gk + (1 - pwr) * gradk * gradk;
        gb = pwr * gb + (1 - pwr) * gradb * gradb;
        cur.first -= (lrk / std::sqrt(gk + dlt))*gradk;
        cur.second -= (lrb / std::sqrt(gb + dlt))*gradb;
        cur_mse = mse(points, cur.first, cur.second);

        way.emplace_back(i, cur.first, cur.second);
        if (std::abs(cur_mse - optimal_mse) < dlt) {
            break;
        }
    }
    return way;
```

Тест на прямой $y = \frac{1}{2}x + 10$

Рис. 9. Ходов: 1105. Время: 9 ms



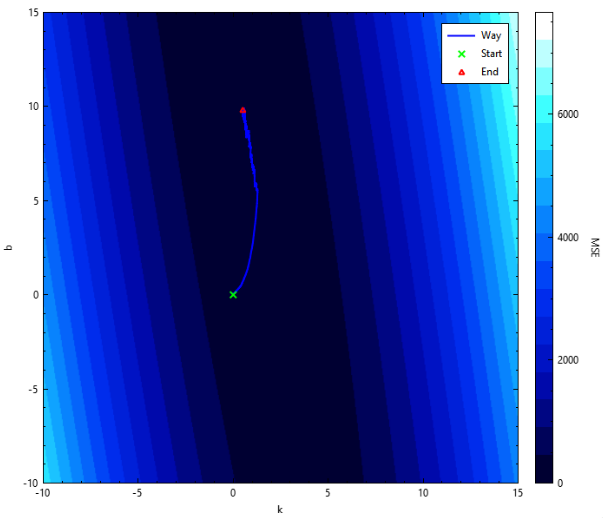
3.5. Ну и наконец-то **Adam**. Adam - сочетание методов Momentum и RMSprop, которое адаптивно обновляет скорость обучения для каждого параметра, используя первый и второй моменты градиента. Это автоматически адаптирует метод к различным условиям оптимизации, учитывая динамику каждого параметра. Adam является эффективным и быстрым методом оптимизации для моделей, включая линейную регрессию.

Ниже представлен код:

```cpp
v<QCPCurveData> adam_linear_regression(const way_t &points, qreal const lrk, qreal const lrb, const qreal k, const qreal b, const int max_step, const qreal dlt)
{
    qreal const optimal_mse = mse(points, k, b);
    qreal const pwr1 = 0.9;
    qreal const pwr2 = 0.98;
    qreal p1k = 0;
    qreal p1b = 0;
    qreal p2k = 0;
    qreal p2b = 0;
    qreal diff;
    qreal gradk;
    qreal gradb;
    qreal cur_mse;

    pr<qreal, qreal> cur = {0, 0};
    auto f = [&cur](qreal const x) {
        return cur.first * x + cur.second;
    };
    auto x = [&points](int const i) { return points[i % points.size()].first;};
    auto y = [&points](int const i) { return points[i % points.size()].second;};

    v<QCPCurveData> way = {{0, cur.first, cur.second}};

    for (int i = 1; i < max_step; ++i) {
        diff = y(i) - f(x(i));
        gradk = -2. * diff * x(i);
        gradb = -2. * diff;
        p1k = pwr1 * p1k + (1 - pwr1) * gradk;
        p1b = pwr1 * p1b + (1 - pwr1) * gradb;
        p2k = pwr2 * p2k + (1 - pwr2) * gradk * gradk;
        p2b = pwr2 * p2b + (1 - pwr2) * gradb * gradb;
        p1k /= 1. - std::pow(pwr1, i);
        p1b /= 1. - std::pow(pwr1, i);
        p2k /= 1. - std::pow(pwr2, i);
        p2b /= 1. - std::pow(pwr2, i);
        cur.first -= (lrk / std::sqrt(p2k + dlt))*gradk;
        cur.second -= (lrb / std::sqrt(p2b + dlt))*gradb;
        cur_mse = mse(points, cur.first, cur.second);

        way.emplace_back(i, cur.first, cur.second);
        if (std::abs(cur_mse - optimal_mse) < dlt) {
            break;
        }
    }
    return way;
}
```

Тест на прямой $y = \frac{1}{2}x + 10$

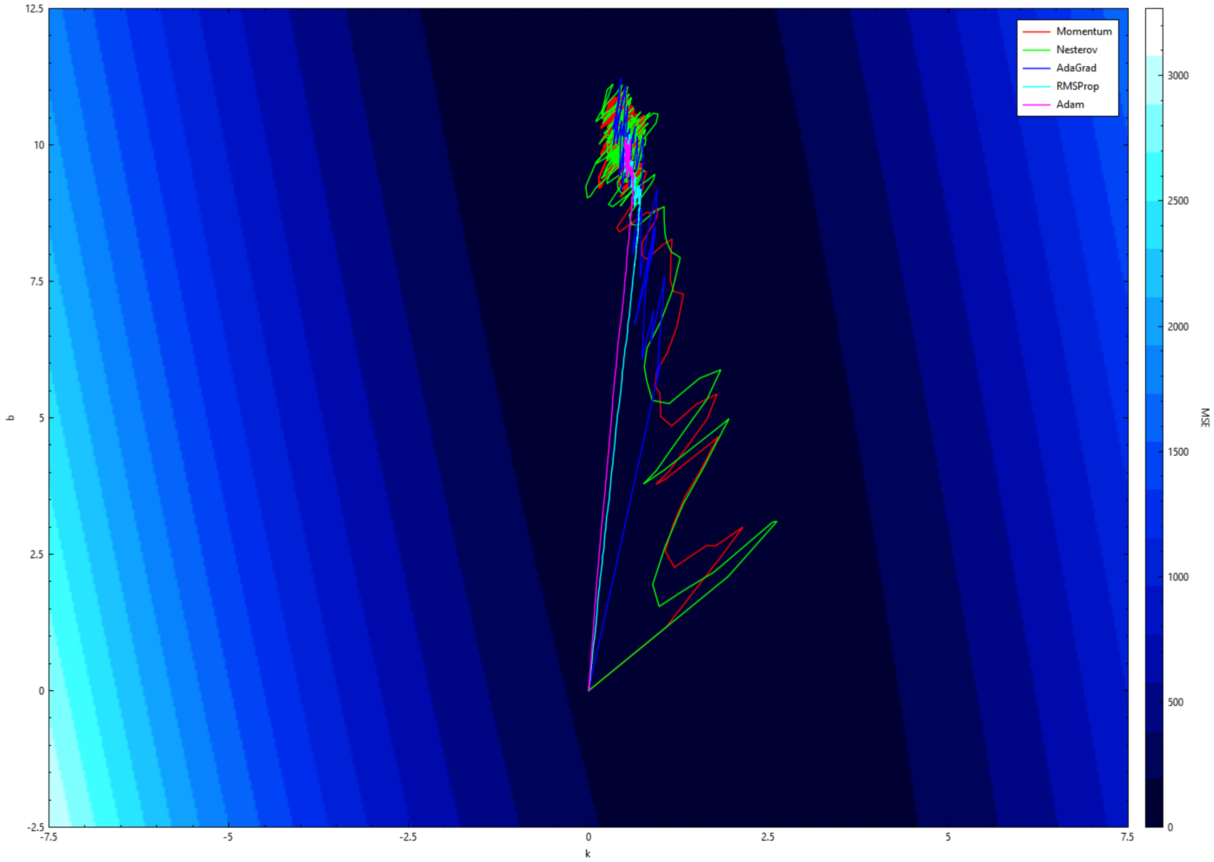
Рис. 10. Ходов: 5096. Время: 41 ms



Итог. Приведём таблицу замеров используемой памяти, времени работы, и т.п.

Метод	Память (%)	Процессор (%)	Сред. время (ms)
GD (batch = 1)	0.1235 (%)	7.8	2
Nesterov	0.2511 (%)	2.87	4
Momentum	0.2398 (%)	3.6	5
RMSPror	0.1733 (%)	4.8	11
AdaGrad	0.1483 (%)	3.06	7
Adam	0.1269 (%)	4.53	39

Ассимтотика у всех алгоритмов одинакова и равна $\mathcal{O}(nm)$, где n - количество точек, а m - количество итераций. Различия могут быть только в константе. Столбцу времени работы в миллисекундах, каждую константу можно спокойно вывести из пропорций.

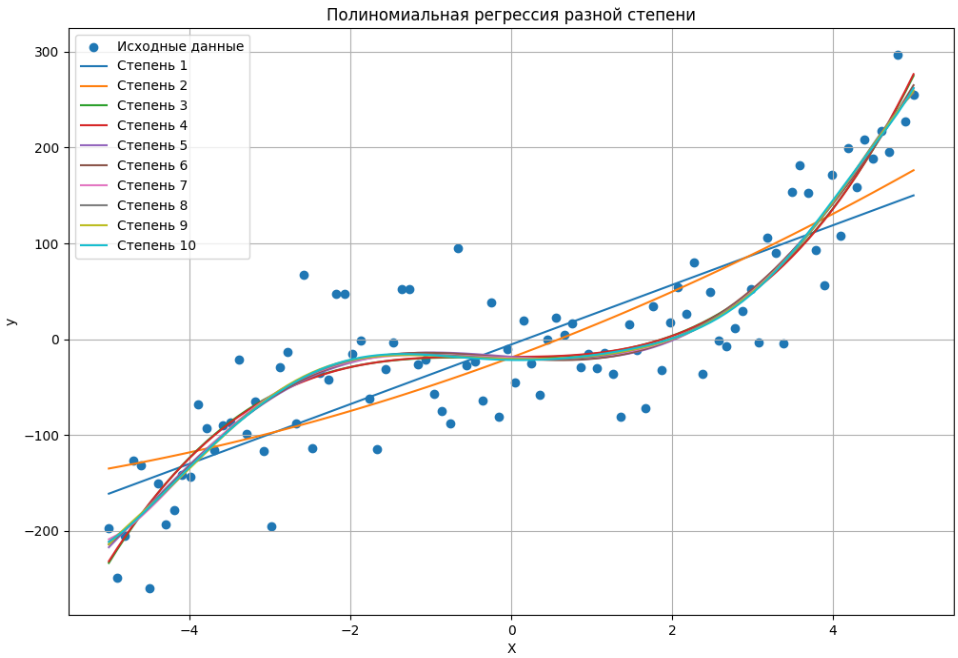


Их этого графика, можно заметить, что рассматривая иетоды мы двигались постепенно в улучшении траектории поиска решения. Например, Momentum имеет гораздо большую дугу чем Adam.

4. **Регрессия на кривых.** Для решения данной задачи, я прибегнул к языку Python, так как там мне показалось удобнее работать с уже готовыми тетодами для работы с матрицами и тд.

   Код:

```python
def poly_reg(x, y, alf=0.11, beta=0.6, deg):
    global result
    A = np.zeros((x.shape[0], deg + 1))
    for i in range(deg + 1):
        A[:, i] = x[:, 0] ** i
    
    result = np.linalg.lstsq(A, y, rcond=None)[0]
    return result
```

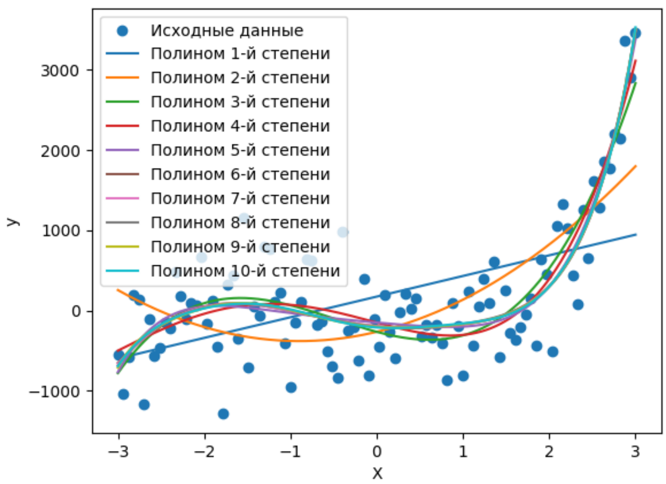


5. С регуляризацией L1

Код:

```python
def poly_reg(x, y, alf=0.11, beta=0.6, deg):
    global result
    A = np.zeros((x.shape[0], deg + 1))
    for i in range(deg + 1):
        A[:, i] = x[:, 0] ** i
    
    result = np.linalg.lstsq(A, y, rcond=None)[0]
    result = np.sign(result) * np.maximum(np.abs(result) - alf, 0)
    return result
```

Пример:

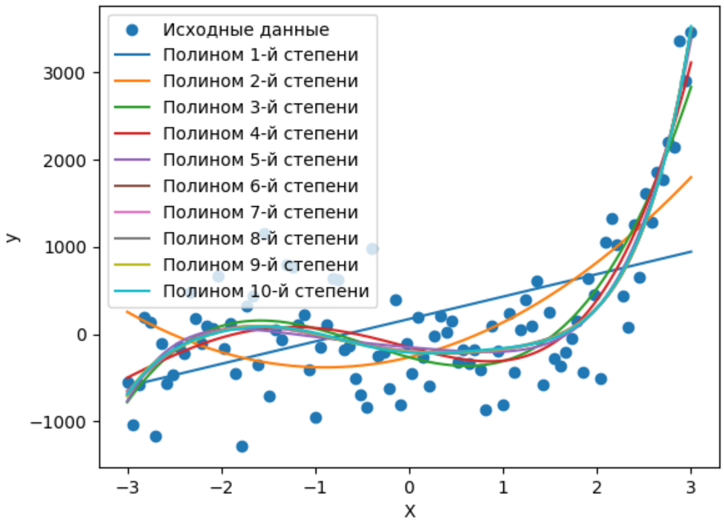


6. С регуляризацией L2

Код:

```
def poly_reg(x, y, alf=0.11, beta=0.6, deg):
    global result
    A = np.zeros((x.shape[0], deg + 1))
    for i in range(deg + 1):
        A[:, i] = x[:, 0] ** i
    
    result = np.linalg.lstsq(A.T.dot(A) + alf * np.eye(deg + 1), A.T.dot(y), rcond=None)[0]
    return result
```

Пример:

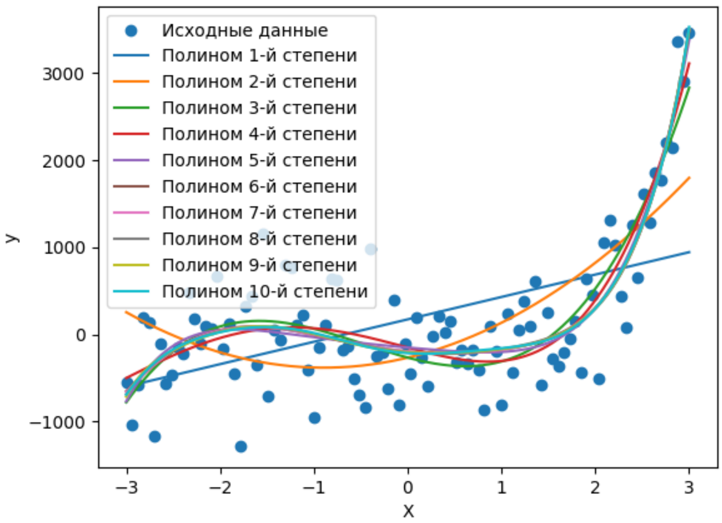


7. С регуляризацией Elastic

   Код:

```
def poly_reg(x, y, alf=0.11, beta=0.6, deg):
    global result
    A = np.zeros((x.shape[0], deg + 1))
    for i in range(deg + 1):
        A[:, i] = x[:, 0] ** i
    
    result = np.linalg.lstsq(
        A.T.dot(A) + alf * (beta * np.eye(deg + 1) + (1 - beta) * np.ones((deg + 1, deg + 1))),
        A.T.dot(y), rcond=None)[0]
    return result
```

   Пример:



Итог

В ходе проведенного исследования мы осуществили глубокий анализ полиномиальной регрессии и метода наименьших квадратов (MSE) с разнообразными модификациями, включая экспоненциальное скользящее среднее (EMA). Мы тщательно изучили и реализовали различные методы для нахождения линейной регрессии, такие как стохастический градиентный спуск (SGD) и его различные варианты (Nesterov, Momentum, AdaGrad, RMSProp, Adam). Кроме того, мы успешно применили полиномиальную регрессию с и без регуляризации, которая способна эффективно устранить резкие перепады в результатах и снизить зависимость регрессии от незначительных аномалий и помех в данных. Это значительно повышает устойчивость и надежность модели.